4-15-1994

Knowledge-Based Nonuniform Crossover

Harpal Maini
Syracuse University, hsmaini@top.cis.syr.edu

Kishan Mehrotra
Syracuse University, mehrotra@syr.edu

Chilukuri K. Mohan
Syracuse University, ckmohan@syr.edu

Sanjay Ranka
Syracuse University

Knowledge-Based Nonuniform Crossover

Harpal Maini *
Kishan Mehrotra
Chilukuri Mohan
Sanjay Ranka
School of Computer and Information Science
4-116 Center for Science and Technology
Syracuse University Syracuse, NY 13244-4100

Email: hsmaini/kishan/mohan/ranka@top.cis.syr.edu
Telephone: 315-443-2368 FAX: 315-443-1122

April 15, 1994

Abstract

We present a new "knowledge-based non-uniform crossover" (KNUX) operator for genetic algorithms (GA's) that generalizes uniform crossover. We extend this to "Dynamic KNUX" (DKNUX), which constantly updates the knowledge extracted so far from the environment's feedback on previously generated chromosomes. KNUX can improve on good solutions previously obtained by using other algorithms. The modifications made by KNUX are orthogonal to other changes in parameters of GA's, and can be pursued together with any other proposed improvements. Whereas most genetic search methods focus on improving the move-selection procedures, after having chosen a fixed move-generation mechanism, KNUX and DKNUX make the move-generation process itself time-dependent. The same parents may give rise to different offspring at different moments in the evolutionary process, based on the past experience of the species. Simulation results show orders of magnitude improvement of KNUX over two-point and uniform crossover, on three NP optimization problems: graph partitioning, soft-decision decoding of linear block codes, and the traveling salesperson problem. KNUX has been applied to variants of the graph partitioning problem that cannot be solved easily using non-GA approaches, and to improve quality of solutions obtained using non-GA methods. DKNUX opens up the field of applying GA's to *Incremental Optimization* problems, characterized by a slow change in problem structure with time. DKNUX also achieves some of the goals of diploid representations with adaptive dominance, with smaller computational requirements.

*Corresponding author. A preliminary version of this paper will be presented at the IEEE International Conference on Evolutionary Computation, Orlando, 1994

1 Introduction

Analogies with natural processes help in understanding complex processes and suggest new methods for solving problems. Genetic algorithms (GA's) and their cousins (evolutionary programming and evolutionary strategies) are stochastic state-space search techniques drawing inspiration from natural evolutionary mechanisms [1, 36]. These methods maintain populations of individuals that represent potential solutions for optimization problems. Various modifications to the basic algorithms differ in the operators used to generate new candidate solutions and guide exploration of different regions of the search space. Traditional GA's are widely applicable weak methods, which do not always perform very well in large instances of NP-complete problems such as graph partitioning, partly because they do not use prior knowledge about the problem at hand. The search space is enormous, and traditional operators often lead to offspring whose performance may be as poor as that of randomly generated individuals.

This paper attempts to fill the need for operators that are general enough to be applicable to many problems, and whose formulation is assured to lead to offspring of high fitness. We attempt to answer the following questions:

- Can problem-specific knowledge be incorporated into GA's following a single general methodology?
- Can the search method be improved by modifying the move generation step itself, rather than the decision-making process that selects each succeeding generation from the previous generation and its offspring?
- Can GA's be efficiently applied to incremental problems, in which small changes occur over time in the problem parameters?

We first present a brief introduction to the traditional genetic algorithm and relevant modifications. New crossover operators (KNUX and DKNUX) are then given, addressing the questions mentioned above. KNUX and DKNUX utilize problem-specific knowledge and knowledge embodied in a previously obtained solution and can improve on good solutions previously obtained by using other algorithms. The modifications made by these operators are orthogonal to other changes in parameters of GA's, and can be pursued together with many other proposed improvements. In other words, the performance of many existing applications can be potentially improved by using an instance of KNUX.

The later sections of this paper show how KNUX and DKNUX can be applied to several NP-complete problems, i.e., decoding, graph partitioning, and traveling salesperson problems. We give simulation results demonstrating the superiority of these operators when compared to alternative crossover operators, and showing how suboptimal solutions obtained by other methods can be improved upon. For instance, the KNUX-based GA for decoding obtains the same quality of solution as the best known alternative algorithm [3] with computational requirements that are an order of magnitude smaller. For graph partitioning, solution quality obtained using a fast but greedy algorithm is improved by using KNUX. Incremental graph partitioning problems are successfully solved using DKNUX. We also solve a variant of the graph partitioning problem for which traditional algorithms are inadequate. For several problems, we illustrate that the performance of KNUX is far superior to that of operators such as two-point crossover. Distributed implementations have yielded near-linear speedups for GA's using KNUX. An analysis

of schema disruption and recombination is also presented, with a variant of Holland's *Schema Theorem*. We discuss the connection with diploid representations (with adaptive dominance) showing that some of their goals can be achieved at a lower expense using DKNUX.

1.1 Genetic Algorithms

The state of the computation is represented in a GA by a population of individuals, each of which represents a candidate solution. The traditional representation is to encode each individual as a bit-string. A "fitness" function reflects the feedback available from the environment regarding the quality of each candidate solution. In each iteration of the GA, the current population produces many offspring, using "crossover" operators. Two parent individuals often give rise to one or two offspring that share aspects of their parents' representations. Components of representations of offspring are also mutated with a small probability, and other operations such as *inversion* may also be applied, to preserve diversity and enable exploration of under-represented regions of the search space. The number of offspring generated by an individual may either be directly proportional to its fitness or may depend on its relative fitness as compared to other individuals in the current generation. Selection mechanisms extract a new generation of individuals from among the previous generation and their (mutated) offspring.

1.2 Why GA's work

Genetic search is accomplished through the implicit discovery of several regions of the search space while only manipulating a few strings. Better individuals generate more offspring, and hence contribute more of their genes to the next generation. Each individual can be considered to belong to many overlapping connected regions in the search space, called *schema*, described as strings over an alphabet that consists of '0', '1', and '*', a third *don't care* symbol. For instance, the schema 1^{**} represents the region of the search space consisting of strings $\{100, 101, 110, 111\}$. Generating an individual 101 can be viewed as simultaneously generating an instance of each of the schema $\{***, 1**, *0*, **1, 10*, 01*, 1*1, 101\}$. In a population of size M comprised of strings of length L each, there are a total of upto $M.2^L$ different strings which represent 3^L schemas of the search space.

Holland showed that the optimum way to explore the search space is to allocate reproductive trials to individuals in proportion to their fitness relative to the rest of the population. In this way, highly fit schemata (whose individual instances have high average fitness) receive an exponentially increasing number of trials in successive generations. Since each individual contains a great many different schemata, the number of schemata which are effectively being processed in each generation is of the order of n^3, where n is the population size. GA's perform well because of this process of simultaneously exploring a large number of schema, a property known as *implicit parallelism*[1]. GA's find good *building blocks*, i.e., high fitness schemata with a small number of defined (0 or 1) bits. If the contribution to overall fitness of each gene were independent of all other genes, then it would be possible to solve the problem by hill climbing on each gene in turn. Unfortunately, this is not the case in real-life problems. A purely hill

[1]Implicit parallelism does not refer to the ability of genetic algorithms to be scalably parallelized on appropriate hardware.

climbing method is good at exploitation (of the knowledge gained by the search conducted so far), but does little exploration of the search space. This is in contrast with a purely random search procedure is good at exploration, but does no exploitation. Holland showed that GA's optimally balance exploration and exploitation for some problems, maximizing expected performance.

1.3 Crossover Operators

The focus of this paper is on crossover, and we now discuss some existing crossover operators, whose choice depends on the application. These are fairly general operators, though more appropriate operators are often used for specific problems where these do not work well.

1. Holland [1] suggested the one-point crossover (1PTX) operator, which is defined in terms of a 'crossover point,' *i.e.,* a place between loci of a chromosome where individuals can be split. Let $\alpha, \beta, \gamma, \delta$ represent strings. One-point crossover works by selecting a point at which chromosomes $\alpha\beta$ and $\gamma\delta$ are cleaved, and the parts recombined to produce $\alpha\delta$ and $\gamma\beta$. It is referred to as one-point crossover since a single site is chosen for splitting the chromosome.

2. Every crossover step "disrupts" a large number of schema, breaking linkages between coadapted alleles of genes in parent individuals, just as it generates instances of a large number of schema. In 1PTX, the probability of disrupting a schema (instantiated by a parent individual) increases with the distance between the first and last defined symbols (1 or 0) in the schema. For instance, 1***0 is far more likely to be broken up than **10*. A reduction in the disruptive effect has been claimed by choosing two crossover points instead of one, as in
$$\alpha_1\beta_1\gamma_1 + \alpha_2\beta_2\gamma_2 \rightarrow \alpha_1\beta_2\gamma_1 + \alpha_2\beta_1\gamma_2.$$
This is known as two-point crossover (2PTX) [30, 20], and often leads to improved performance.

3. A natural generalization of two-point crossover is k-point crossover (kPTX) where $k \geq 1$ is the number of points at which parent strings are broken.
$$\alpha_0\alpha_1 \ldots \alpha_k + \beta_0\beta_1 \ldots \beta_k \rightarrow \alpha_0\beta_1\alpha_2\beta_3 \ldots + \beta_0\alpha_1\beta_2\alpha_3 \ldots$$

4. k-point crossover can be understood in terms of a "crossover mask" [14] in which there are $k+1$ contiguous segments of identical bits, and the boundary between adjacent different-valued segments represents a crossover point. The i^{th} bit of the offspring is inherited from one parent if the i^{th} bit in the mask is 0, and from the second parent otherwise. For instance, the mask 000011110000 represents a two-point crossover with the crossover points being positions 4 and 8. Uniform crossover (UX) [14] generalizes the crossover mask to be any bit string (of the chromosomal length), where each bit in the crossover mask is equally likely to be 1 or 0. If a particular uniform crossover application was such that only the first, third, fourth, and last bits were inherited from the first parent, the corresponding mask is 010011111110, which can also be viewed as 4-point crossover at positions 1,2,4, and 11. On the average, uniform crossover produces $L/2$ crossings on a string of length L [32].

5. Variations of UX include "HUX" [16], where the crossover mask is enforced to contain roughly the same numbers of 1's and 0's. Spears and De Jong suggested a parameterized uniform crossover where every bit is selected from either parent with a fixed probability p_0 [32].

   Syswerda [14] defined recombination potential as the ability of crossover to create higher order hyperplanes when the parents contain the necessary lower order hyperplanes. He provided an analysis showing uniform crossover to have a higher recombination potential than 1-point and 2-point crossover. Spears and De Jong [32], have extended this analysis to p_0-uniform crossover. They point out .5 uniform crossover is better at schema recombination but is worse for schema survival, whereas .1 uniform crossover is worse for schema recombination but better for schema survival. With $p_0 = 0.1$, uniform crossover is less disruptive than 2-point crossover while at the same time not having a defining length bias. This allows uniform crossover to perform equally well, regardless of the distribution of important alleles. They find that for a population of size 1000, uniform crossover with $p_0 = 0.2$ outperforms 2-point crossover.

1.4 GA's That Use Problem-Specific Knowledge

The following techniques for using problem-specific knowledge have been suggested in the literature. In later sections, we propose a more general crossover operator and framework that accomplishes this.

Grefenstette argues in [13] that it is possible to exploit problem-specific knowledge in almost every phase of a genetic algorithm. Seeding the initial population with a heuristic solution, local search coupled with mutation and problem specific recombination operators are suggested mechanisms. Grefenstette solved the Traveling Salesperson Problem (TSP) by using the cost of tour edges in the construction of offspring.

Suh and Van Gucht [33] say that the choice of representation for an optimization problem is interlinked with the choice of crossover operator. In particular, the representation should allow for the definition of recombination operators that incorporate heuristic information about the problem. In combinatorial optimization, an annealing-like approach that uses distance information can enhance the performance of recombination operators on the TSP [33].

Louis and Rawlins suggest exploiting knowledge of the search space topography in guiding genetic search [34]. They use the fitness difference between parents and children to indicate directions to bias search, advocating this as a useful computing technique although such directional information cannot be explicitly stored and used in nature. They define a Masked Crossover (MX) that uses the relative fitness of the children with respect to their parents. A mask is associated with each parent, and each offspring is produced by a parent dominating the crossover. A set of rules operate for each bit, controlling future settings on masks. They collect runtime statistics to identify deception using a modification of the ANODE algorithm suggested by Goldberg [7], [8].

2 Knowledge-Based Nonuniform Crossover

We develop KNUX, a crossover operator that effectively exploits knowledge inherent in the problem and problem-specific information provided by the history of the genetic search process. The uniform crossover operator (UX) selects either bit of the parent with equal probability. This ignores the fact that one parent may have much higher fitness than another, or that one region of the search space is already known to produce individuals of higher fitness than other regions. KNUX rectifies this situation, generalizing UX to make use of the knowledge of fitness of various individuals already explored. Problem-dependent knowledge can be garnered while genetic search is in progress. The best solution found up to a certain point can be used to build the bias vector. This gives us a dynamic version of KNUX, referred to as DKNUX (Dynamic KNUX). We obtain orders of magnitude improvement over traditional genetic techniques without an increase in computational cost.

UX can be described in terms of a bit-vector mask, each bit of which determines the parent from which an offspring inherits a value for a particular bit-position. Generalizing this idea, KNUX uses a crossover mask $\boldsymbol{p} = (p_1, \ldots, p_n)$, where each p_i is a real number $\in [0, 1]$ (p_i's are not necessarily equal). Each member of the string denotes a bias (probability) towards selecting genetic material from one parent, or the bias towards selecting a bit-value that equals the allele in the corresponding bit-position in a special currently known reference vector. Bias probability depends on the position of the bit in the string, the relative fitness of the parent strings, and on problem-specific knowledge. Given $\boldsymbol{p}$ and the two parents, $\boldsymbol{a} = (a_1, \ldots, a_n)$ and $\boldsymbol{b} = (b_1, \ldots, b_n)$, the offspring $\boldsymbol{c} = (c_1, \ldots, c_n)$ is obtained such that if $a_i = b_i$, then $c_i = a_i$, else the probability that $c_i = a_i$ is p_i. Allowing $\boldsymbol{p}$ to vary dynamically as the search progresses gives us operator DKNUX. In the following sections we describe a distributed implementation of a GA using KNUX, and then show how to build bias probabilities p_i for the problems of graph partitioning, soft-decision decoding of linear block codes and the traveling salesperson problem.

The fittest individual(s) in a population contain certain properties that make them perform better relative to others. This means that certain features of these individuals can be usefully emulated by others in the population. Using this rationale, DKNUX dynamically exploits the performance of the best individual found in the search process upto a certain time. In the graph partitioning problem, for example, this is done by rebuilding the appropriate probability distribution that reflects the neighborhood locality relation of a good solution.

2.1 Distributed Population Model

Several researchers have made a very strong case for distributed genetic algorithms with advantages ranging from linear scalable speedups to improved solution quality due to niching [6, 10, 9, 12]. The results of a study conducted by Collins and Jefferson [11] indicates that local mating is more appropriate for artificial evolution than *panmictic* mating, where any individual can mate with any other. It has been observed that genetic algorithms typically converge on a single peak of a multi-modal objective function when several solutions of equal quality exist. Panmictic selection-based genetic algorithms focus on one or more optimal solutions in the early generations and eventually converge on or near that solution, whereas local mating allows convergence towards more than one solution.

We use a coarse-grained, distributed-population genetic algorithm (DPGA). In this model,

the individuals are divided into a number of subpopulations that exchange genetic information through an explicit exchange of individuals; mates are selected for reproduction from the local subpopulation.

The algorithm described in Figure 1 implements the proposed coarse-grained distributed-population genetic algorithm. The algorithm requires as input *NumberSubpopulations* (the number of subpopulations), *InterconnectionMatrix* (which indicates the neighborhood relation between subpopulations) and *ParamsMatrix* which holds the size of the subpopulation, *pcross* (the crossover probability), *pmut* (the mutation probability), *MigrationIntensity* (the migration intensity), *MigrationInterval* (the migration interval at each subpopulation), and *DKNUXInterval* (the interval at which probability distributions are rebuilt). The following notation is used:

- p_m = probability of mutating a single bit, chosen to be 0.01 in our experiments;
- p_{cross} = the crossover probability, chosen to be 0.7 in our experiments;
- N = size of population, chosen to be 320, 300 or 128 in our experiments;
- $rank(\boldsymbol{x}, P) = |\{\ \boldsymbol{y} \in P : fitness(\boldsymbol{y}) \geq fitness(\ \boldsymbol{x})\}|$ where P is a population that contains $\boldsymbol{x}$; and
- $virility(\boldsymbol{x}, P) = max_v - \Delta_v.rank(\boldsymbol{x}, P)$, with $max_v = 2.4$ and $\Delta_v = 0.8/N$ in experiments.

Time Complexity of DPGA

Let N = Total population size, s = subpopulation size, t = number of subpopulations, i = number of immigrants arriving at a subpopulation, n = number of locii in each chromosome, b = number of boundary points in a partition. The time complexity of selection and ranking is seen to be $O((s+i)\log(s+i))$ at each subpopulation, which is $O(t(s+i)\log(s+i))$ for all t subpopulations. This reduces to $O(N\log(s+i))$. Let $O(F)$ be the cost of fitness computation. This means the overall time complexity of algorithm DPGA is $O(N\log(s+i) + NF)$ for each generation.

3 Applications of KNUX and DKNUX for Optimization

In this section, we present applications of KNUX and DKNUX to three NP optimization problems. These are graph partitioning, soft-decision decoding of linear block codes, and the Traveling Salesperson Problem. A more detailed description of the former may be found in [27] and [37].

Algorithm DPGA (*NumberSubpopulations*, *InterconnectionMatrix*, *ParamsMatrix*)

- Create a number of subpopulations linked by an interconnection matrix.
- The subpopulations are initialized randomly or alternatively contain an estimate of a good solution.
- if (DKNUXInterval) rebuild the KNUX probability distribution.
- At all subpopulations do

while (*generation-counter* $<$ *Total-Number-Generations*) do

while (MigrationInterval)

Transfer selected immigrants to immediate neighboring subpopulations.

Gather immigrants and individuals into mating pool.

Sort and rank individuals in the current subpopulation mating pool.

Allocate reproductive trials to individuals in mating pool such that the expected number of crossover operations in which an individual participates = $virility(\boldsymbol{x}, P)$.

while (*population-size-counter* $< N$) do

Randomly select two individuals for reproduction.

If *random-num* $\in [0,1] >$ *pcross*, then introduce both individuals into the next generation with equal probability, else (i) apply KNUX to $\boldsymbol{a}$ and $\boldsymbol{b}$, producing offspring $\boldsymbol{c}$. (ii) Mutate $\boldsymbol{c}$. (iii) Perform hill-climbing on the resultant mutant. (iv) Introduce the resulting individual into the next generation.

end while

end while

Figure 1: Distributed Population Genetic Algorithm

3.1 Graph Partitioning

Fast solutions for the graph partitioning problem are extremely important in parallel computing and research areas such as circuit partitioning for VLSI design. For instance, parallelization of many scientific and engineering problems requires partitioning the data among the processors in such a fashion that the computation load on each node is balanced, while communication is minimized. This is a graph partitioning problem, where nodes of the graph represent computational tasks, and edges describe the communication between tasks; each partition corresponds to one processor. Optimal partitioning, if possible, would allow optimal parallelization of the computations, with load balanced over various processors, and with communication time minimized. The computational graph often has the property that adjacent vertices are physically proximate, and vice versa. For many applications, the computational graph can be derived only at runtime, and requires that graph partitioning also be done in parallel.

The problem may be formally stated as follows. Consider a graph $G = (V, E)$, where V represents a set of vertices, E represents a set of undirected edges, the number of vertices is given by $n = |V|$, and the number of edges is given by $m = |E|$. The graph partitioning problem can be defined as an assignment scheme $M : V \longrightarrow P$ that maps vertices to partitions. We denote by $B(q)$ the set of vertices assigned to a partition q, *i.e.*, $B(q) = \{v \in V : M(v) = q\}$. For graphs representing the computational structure of a physical domain, each vertex $v_i \in V$, $1 \leq i \leq m$, corresponds to a point in d-dimensional space with physical coordinates $(x_{i_1}, x_{i_2}, \ldots, x_{i_d})$, and each edge is an ordered pair (v_i, v_j) that may connect physically proximate vertices.

The weight w_i corresponds to the computation cost (or weight) of the vertex v_i. The cost of an edge $w_e(v_1, v_2)$ is given by the amount of interaction between vertices v_1 and v_2. Thus the weight of every partition can be defined as

$$W(q) = \sum_{v_i \in B(q)} w_i.$$

The cost of all the outgoing edges from a partition represents the total amount of communication cost and is given by

$$C(q) = \sum_{v_i \in B(q), v_j \notin B(q)} w_e(v_i, v_j) + \sum_{v_j \in B(q), v_i \notin B(q)} w_e(v_j, v_i).$$

We would like to make an assignment such that the total computation and communication time $(W(q) + \beta C(q))$ spent by every node is minimized, where β represents the ratio of the cost of unit computation to cost of unit communication on a given machine.[2]

Like all good things in life, graph partitioning is NP-complete [24]. Optimally bipartitioning a 1000-node graph may require examining a search space of $O(2^{1000})$, and no machine would wait to obtain such a solution. Obtaining suboptimal solutions quickly is desirable and often satisfactory. The following describes a knowledge-based approach for graph partitioning:

1. We obtain the vector $\boldsymbol{p}$ by first generating an initial candidate solution. From this initial solution, locality information about the graph can be used to associate a probability with each component in the crossover mask. The initial solution can be obtained by any reasonable heuristic such as the Index Based Partitioner (Appendix A), if coordinate information is available. In the absence of coordinate information, it is possible to use breadth-first

[2]Parameter β cannot be done away with, since optimality of performance depends on the relative costs of communication and computation, which may vary between different machines.

search techniques such as the "graph-layering" heuristic discussed in [29]. The DKNUX operator uses a random or other good initial solution, dynamically changing the vector $\boldsymbol{p}$ as the algorithm steps through successive generations; this allows use of successively better solutions to bias the crossover mechanism.

2. Each node should be biased to be mapped to the same partition as most of its neighbors. Bias probabilities are generated in such a way that each p_i represents the fraction of nodes in the neighborhood of the i^{th} node that belonging to the partition q in a reference solution. In general, the partition produced by the initial solution (or reference vector) can be used to estimate the nature of the neighborhood of a node. Algorithm BUILD_BIAS describes in detail the procedure used to develop bias probabilities, taking as input the *graph* to be partitioned, the *no_of_partitions* required and the *no_of_nodes* in the graph. Algorithm KNUX_GRAPH shows how the bias matrix thus constructed can be used to implement knowledge-based crossover; inputs to this procedure are the number of nodes in the graph to be partitioned, the bias matrix $\boldsymbol{P}$ and parents $\boldsymbol{a}$ and $\boldsymbol{b}$ to which crossover must be applied. The (i,j)th component of the bias matrix represents the fraction of nearest neighbors of node i that are assigned to partition j.

   Figure 2 illustrates the process of building a bias vector for a 16 node graph and the resulting effect on KNUX.

3.1.1 Simulation Results

In this section, we provide a comparison between different crossover operators in algorithm DPGA; two-point (TPTX), the knowledge-based nonuniform crossover (KNUX), and the dynamic knowledge-based nonuniform crossover (DKNUX).

The results obtained by partitioning several different graphs into different sized partitions are presented in [37]. Table 1 and Figure 5 exhibit the results of partitioning graphs of 66 and 258 nodes (given in Appendix B) into 2, 4 and 8 partitions. Figure 5 is obtained by averaging the results of 5 runs of algorithm DPGA. Table 1 represent the best solutions obtained in these 5 runs and these are compared with solutions obtained with recursive spectral bisection. All experiments were done using algorithm DPGA, with a total population size of 320, a crossover rate of $p_c = 0.7$ and the mutation rate of $p_m = 0.01$. The experiments were done with two different subpopulation configurations in algorithm DPGA; 16 subpopulations configured as a four dimensional hypercube and a single population.

The results establish very clearly the excellent performance of KNUX and DKNUX in comparison with two-point crossover and also that DKNUX is competitive with recursive spectral bisection as a graph partitioning strategy. These methods are discussed in detail in the rest of this section.

In Table 1, we also show a comparison of the best solution found using Recursive Spectral Bisection and that found using DKNUX in algorithm DPGA. For the 66 node graph, we find that DPGA with operator DKNUX outperforms recursive spectral bisection in terms of both the best and average solutions found. DKNUX outperforms recursive spectral bisection in under 50 generations for two-way and four-way partitioning, and does so in about 150 generations for the eight-way partitioning case. In the case of the 258 node graph, the performance of DKNUX was not as good as that of recursive spectral bisection. This was alleviated by seeding the

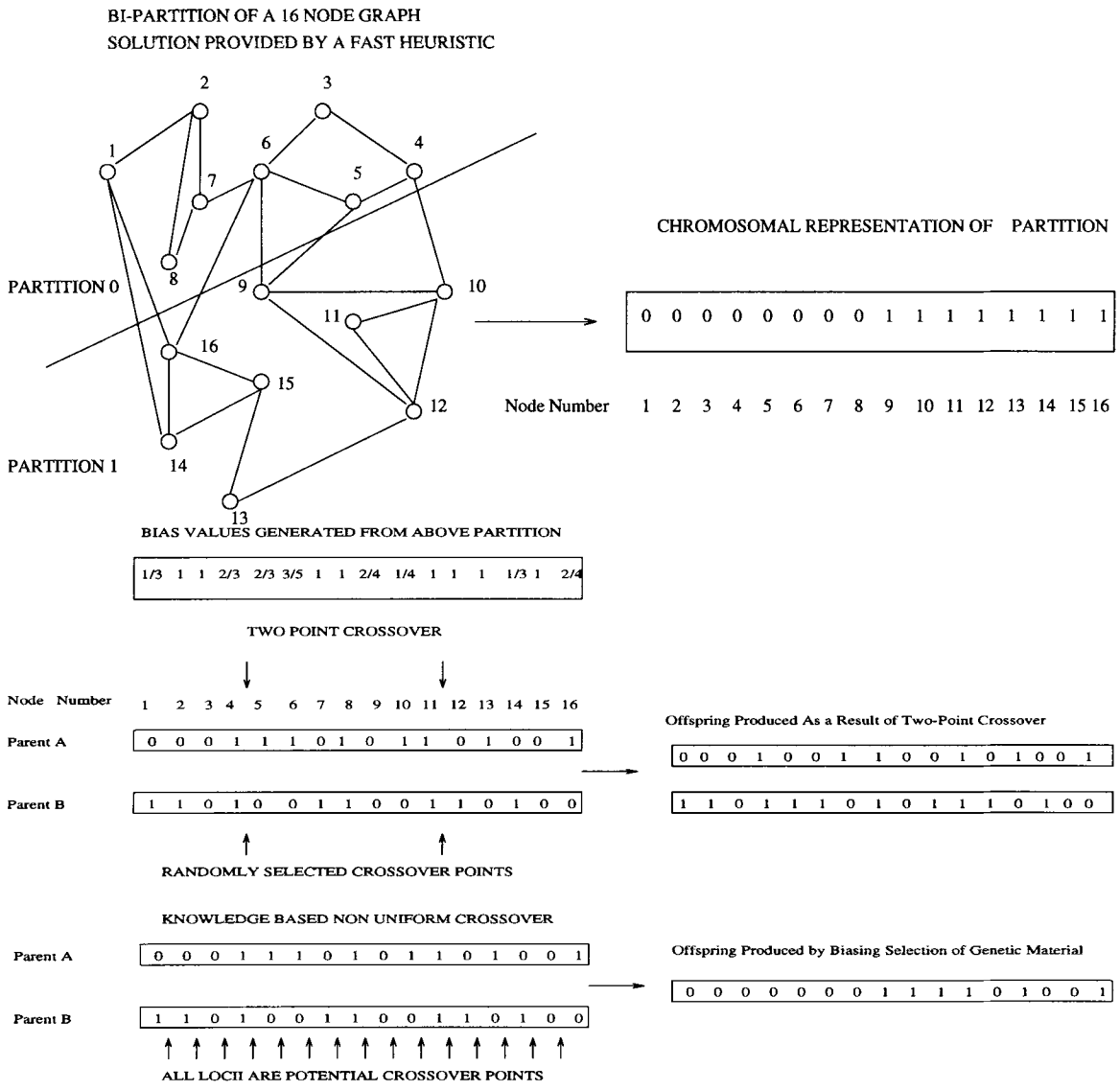

Figure 2: An Example of Bias Probability Generation & Crossover Operator KNUX for Graph Partitioning

Algorithm BUILD_BIAS(*graph*, *no_of_partitions*, *no_of_nodes*)

- for all nodes in the *graph* do
  - for all immediate neighbors of current node
  - find partitions to which all immediate neighbors of current node are assigned
  - build bias vector with *no_of_partitions* components with each component p_i representing the fraction of neighbors assigned to partition i
- return bias matrix $\boldsymbol{P}$ which has *no_of_nodes* rows and *no_of_partitions* columns. Each row of the bias matrix is the bias vector, found for the corresponding node.

Figure 3: Algorithm for Building Bias Probabilities

Operator KNUX_GRAPH(*no_of_nodes*, $\boldsymbol{P}, \boldsymbol{a}, \boldsymbol{b}$)

- for $i := 1$ to *no_of_nodes* do
  - if ($a_i = b_i$) then $p_i = 1.0$,
  - else if $\boldsymbol{P}[i, b_i] + \boldsymbol{P}[i, a_i] = 0$ then $p_i = 0.5$,
  - else $p_i = \frac{\boldsymbol{P}_{[i,a_i]}}{\boldsymbol{P}_{[i,b_i]}+\boldsymbol{P}_{[i,a_i]}}$;
  - set $c_i = a_i$ with probability p_i and $c_i = b_i$ with probability $1 - p_i$.
- return offspring $\boldsymbol{c}$.

Figure 4: Crossover Operator KNUX for Graph Partitioning

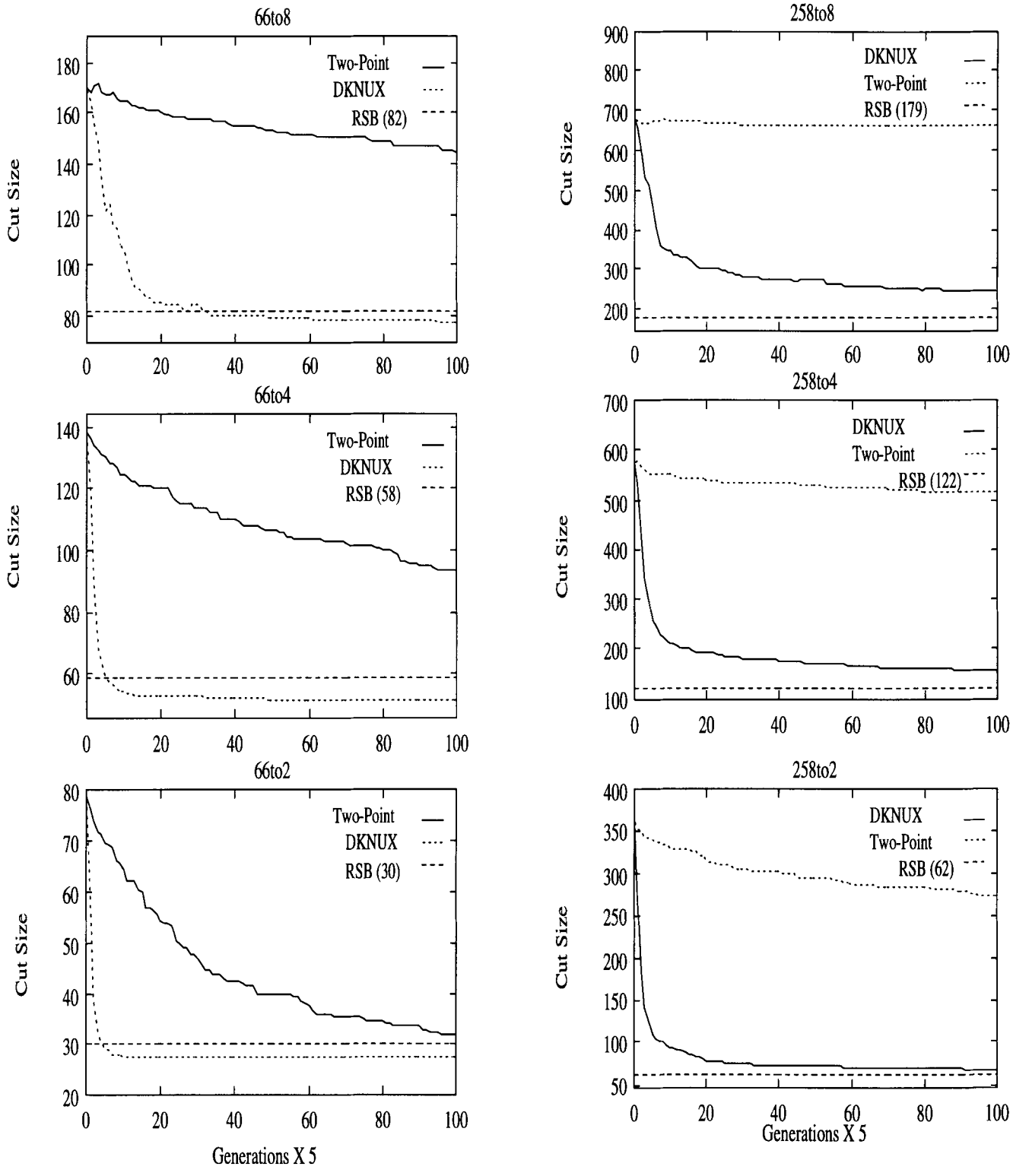

Figure 5: Partitioning 258 node and 66 node graphs into 2, 4 and 8 partitions starting with a randomly initialized population: Comparison between Two-Point and DKNUX.

Number of Partitions	2	4	8
258 Nodes Best Cut Using DKNUX	58	132	216
Cut Using RSB	62	122	179
66 Nodes Best Cut Using DKNUX	26	49	75
Cut Using RSB	30	58	82

Table 1: A Comparison of the Best Solutions found Using DKNUX and RSB: starting with a randomly initialized population.

intial population with a solution obtained through a fast heuristic. Performance was further improved by incorporating a hill-climbing step in algorithm DPGA [37]. This example reminds us that GA's can also get stuck in local minima, as do other stochastic optimization techniques. Increasing the population size may alleviate this problem to some extent.

Genetic algorithms have been used in the past to find good suboptimal solutions to the graph partitioning problem [12, 25, 26, 28]. Exact comparisons of the different algorithms are not available due to the nonexistence of benchmark problems and results. However, our experiments with the traditional crossover operators used by some of these researchers gave results of lower quality than using the operators presented in this paper. Further, the results achieved by our methods are better or comparable to the best known methods for graph partitioning.

3.2 Soft Decision Decoding of Linear Block Codes

In this section we describe the problem of soft-decision decoding of linear block codes and the role of KNUX in this problem.

Codes are used for the reliable transmission of data over communication channels susceptible to noise. Codes may be classified as either block codes or tree codes. An encoder for a block code accepts as input a k symbol message sequence (usually binary sequence) and maps it to an n ($> k$) symbol sequence. Each n-symbol sequence is completely determined by a specific k-symbol message. Block codes may further be classified as linear or nonlinear. A linear code is defined as a vector space over a finite field. We restrict our attention to binary linear codes. Figure 6 describes a typical communication system. As a result of noise, the received vector components are real numbers. Of the n codeword coordinates, exactly k are linearly independent. Let $\boldsymbol{i}$ be the information vector and $\mathbf{G} = (g_{jm})$ the generator matrix, a listing of the basis vectors of a code C; then the encoding operation yields $\boldsymbol{i}\mathbf{G} = \boldsymbol{c}$, and, consequently, $c_j = \sum_{m=1}^{k} i_m g_{jm}$ represents the j-th component of the codeword, $\boldsymbol{c}$. Let $\boldsymbol{r}$ be a received vector $\in \Re^n$. "Hard decision" decoding involves quantizing each component of the received vector independently to the nearest value $\in \{-1, 1\}$ and then moving to the code-vector nearest to the resulting sequence. "Soft-decision" decoding algorithms utilize received vector components, not just their quantized estimates [5]. A maximum-likelihood decoder finds a codeword $\boldsymbol{c}'$ that maximizes the conditional probability of receiving $\boldsymbol{r}$, i.e.

$$P(\boldsymbol{c}'|\boldsymbol{r}) = \max_{\boldsymbol{c}} P(\boldsymbol{c}|\boldsymbol{r}) = \max_{\boldsymbol{c}\in C} P(\boldsymbol{r}|\boldsymbol{c})P(\boldsymbol{c})/P(\boldsymbol{r})$$

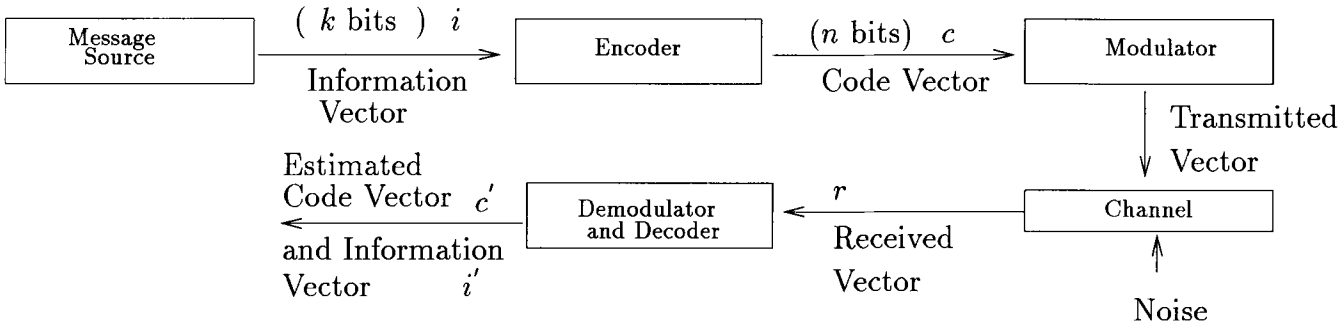

Figure 6: A Typical Communication System

The above equation holds since we assume that all codewords are equally likely to be transmitted. A maximum-likelihood decoder is optimal in this sense. If transmitted signals are binary antipodal ($\in \{-1,1\}^n$) over a discrete memoryless channel susceptible to additive white Gaussian noise, and the noise affects each symbol independently, then $P(\boldsymbol{r}|\boldsymbol{c}')$ is maximized when the squared Euclidean distance between vector $\boldsymbol{r}$ and $\boldsymbol{c}'$, $\sum_{j=1}^{n}(r_j - c'_j)^2$, is minimized [2], [4]. Thus maximum-likelihood decoding reduces to nearest-neighbor decoding, with the Euclidean metric. More formally, the soft-decision decoding problem reduces to the following:
Given received real vector $\boldsymbol{r} = (r_1, ..., r_n)$, find a codeword $\boldsymbol{c} \in C$ that minimizes $\sum_{j=1}^{n}(r_j - c_j)^2$.

Most research in decoding algorithms has been focused on hard-decision decoding algorithms based on algebraic techniques. Soft-decision decoding has not been as extensively studied and until recently there were not many efficient decoding algorithms for linear block codes of large block length. The problem of decoding an error correcting code is known to be NP-hard. It is indeed desirable and often preferable to obtain suboptimal solutions to such a problem. To date, the recently developed A^*-based decoding algorithm [3], is probably the most successful algorithm for soft decision decoding. We have been able to obtain results of the same quality using an instance of KNUX, with an order of magnitude lower computational requirements.

3.2.1 GA for Soft-Decision Decoding

In this section, we present an instance of algorithm DPGA, configured as a single panmictic population, referred to as GADEC [27], that performs soft-decision decoding of linear block codes.

Notation:

n = blocklength, ranging up to 104 in our experiments to date,
k = dimension of the linear code, ranging up to 52 so far,
Y = signal to noise ratio (decibels), varied betwen 1.5dB to 2.5dB in our experiments,
$\boldsymbol{r}$ = received vector $\in \Re^n$,
$\boldsymbol{r}'$ = the result of permuting $\boldsymbol{r}$ so that the first k positions are the most reliable (high magnitude) linearly independent positions of $\boldsymbol{r}$,
$\boldsymbol{h}=\{sgn(r'_1), \ldots, sgn(r'_k)\} \quad \in \{-1,1\}^k$
$fitness(\boldsymbol{x}) = -\sum_{i=1}^{n}(r'_i - coded(x)_i)^2$, the negated Euclidean distance,

Components of Permuted Received Vector ($\boldsymbol{r}'$)	-2.1	+1.8	0.9	-0.7
Bias Probability ($\frac{1}{1+\exp\frac{-2r_i'}{\sigma^2}}$)	.0012	0.99	0.94	.09

Table 2: An Example of Bias Probability Generation at a Signal-to-Noise Ratio of 2.0 dB for a Code of Dimension 4.

Operator KNUX_DECODE($\boldsymbol{a}$,$\boldsymbol{b}$,$\boldsymbol{r}'$)

- for i := 1 to k do
  - Let the i^{th} component of the offspring be either 1 or -1, with

$$P(c_i = 1) = \begin{cases} 0 & \text{if } a_i = b_i = -1 \\ 1 & \text{if } a_i = b_i \\ \frac{1}{1+\exp\frac{-2r_i'}{\sigma^2}} & \text{if } a_i \neq b_i \end{cases}$$

$$\text{and} \quad P(c_i = -1) = 1 - P(c_i = 1)$$

- return offspring c.

Figure 7: Crossover Operator KNUX for Soft-Decision Decoding

The main features of the algorithm are highlighted below:

1. The initial population contains $\boldsymbol{h}$ and also $(N-1)$ randomly generated vectors $\in \{-1, 1\}^k$.
2. Crossover and mutation operate only on the information bits, which means that all the individuals in the population are always feasible solutions.
3. Since received vector components with greater magnitudes are more reliable [3], the received vector is reordered so that the most reliable k linearly independent bits come first. A choice between inheriting a bit from either of two parents is made at each component of an offspring using operator KNUX_DECODE (Figure 7), and the location of bits on the problem encoding is irrelevant, resulting in better performance than 'one-point' and 'two-point' crossover operators [14], [15], [16].
4. Bias probabilities for operator KNUX are determined by the components of the permuted received vector $\boldsymbol{r}'$. Algorithm KNUX_DECODE shows how the bias values thus constructed can be used to implement knowledge-based crossover by computing $\frac{1}{1+\exp\frac{-2r_i'}{\sigma^2}}$. Table 2 gives an example of using $\boldsymbol{r}'$ to compute bias probabilities for a code with $k = 4$.
5. Genetic search is guided by the principle of increasing fitness, i.e., decreasing Euclidean distance to the received vector.

6. The selection strategy used is "Linear Ranking" [19], shown to result in accurate optimization, avoiding local minima [18]. The relative (fitness-dependent) rank of each individual determines the number of reproductive trials in which the individual participates.

7. Only the fitter of two possible offspring is preserved; also, the algorithm explicitly enforces the survival of the best individual in the current generation into the next generation. Conservative selection policies and uniform crossover coupled with a technique for exploiting problem specific knowledge lead to very good performance.

3.2.2 Simulation Results and Discussion

We present simulation results at various signal-to-noise ratios for the [104, 52] extended binary quadratic residue code. This is a large code, with a search space of size 2^{52}. Results presented in Figure 8 and Table 3 are for a single population of size 300, a crossover rate of 70% and a mutation rate of 3%.

Although the genetic algorithm is used to minimize the squared Euclidean distance between vector $\boldsymbol{r}$ and $\boldsymbol{c}'$, $\sum_{j=1}^{n}(r_j - c'_j)^2$, the overall performance of decoders is measured in terms of the bit error probability. The bit error probability (P_b) is defined as the average fraction of information bits in error upon decoding. We present graphs and data to illustrate the relation between P_b and the number of genetic algorithm generations. P_b is estimated by simulating the transmission of several codewords (about 1000) at many different signal-to-noise ratios.

Figure 8 indicates that bit error probability drops rapidly for the first five generations, then settles into a more steady rate of improvement as shown in Figure 8. Bit-error-probability continues to decrease with increasing number of generations, though at a slower rate; it is possible to balance solution quality with computational efficiency by choosing a combination of the solution quality and the maximum number of generations as termination criterion for the GA. When the number of generations is increased from 50 to 100, for instance, bit error probability drops by about 30% to 40%.

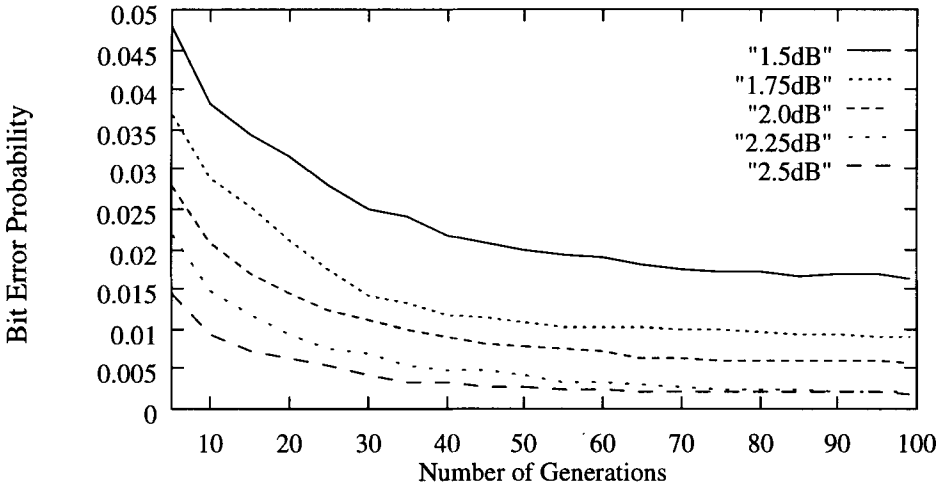

Figure 8: Bit Error Probability in the Later Stages of Evolution

In Table 3, we present bit error probability and related statistics after 50 and 100 genetic algorithm generations. Figures 9 and 10 exhibit the relation between bit-error probability and

Signal-to-Noise Ratio, dB	1.5	1.75	2.0	2.25	2.5
Bit-Error Probability (Uncoded Data)	.0462	.0418	.0375	.0334	.0296
No. of Codewords Evaluated	30000	30000	30000	30000	30000
Number of Generations	100	100	100	100	100
P_b, Bit-Error Probability (Coded Data)	0.0165	0.00873	0.00563	0.00217	0.00183
MLD Lower Bound	0.00904	0.00404	0.00267	0.00135	0.00083
Ratio, P_b/MLD	1.82	2.16	2.10	1.65	2.2
No. of Codewords Evaluated	15000	15000	15000	15000	15000
Number of Generations	50	50	50	50	50
P_b, Bit-Error Probability (Coded Data)	0.019730	0.010838	0.007769	0.004200	0.002569
MLD Lower Bound	0.008307	0.004039	0.002676	0.001336	0.000810
Ratio, P_b/MLD	2.37	2.28	2.90	3.14	3.17

Table 3: Simulation Results using GA with KNUX for the $[104, 52]$ Code.

the signal-to-noise ratio, after 50 and 100 generations, respectively. Some of these errors would necessarily be made by any maximum likelihood decoder as well and reflect cases where a codeword other than the transmitted codeword was found to be closer to the received vector, $\boldsymbol{r}'$.[3] This "lower bound" is also presented. The average fraction of bits in error in these cases is referred to as MLD in Table 3. The difference between the two curves, given in Figures 9 and 10, is often used to gauge the performance of a suboptimal decoding algorithm.

Remarkable results are obtained using **GADEC**, as seen by examining the ratio of the bit

[3]For instance, there is a finite probability that all the message bits are so severely corrupted by the communication channel that even a theoretical maximum likelihood decoder will recover a codeword different from the transmitted codeword.

Signal-to-Noise Ratio, dB	1.5	2.0	2.5
No. of Codewords Evaluated	30000	30000	30000
Number of Generations	100	100	100
P_b, Bit-Error Probability, KNUX (Coded Data)	0.0165	0.00563	0.00183
P_b, Bit-Error Probability, UX (Coded Data)	0.151	0.116	0.0851
P_b, Bit-Error Probability, 2PTX (Coded Data)	0.130	0.098	0.065

Table 4: Comparison Between Different Crossover Operators for GA applied to Soft-Decision Decoding

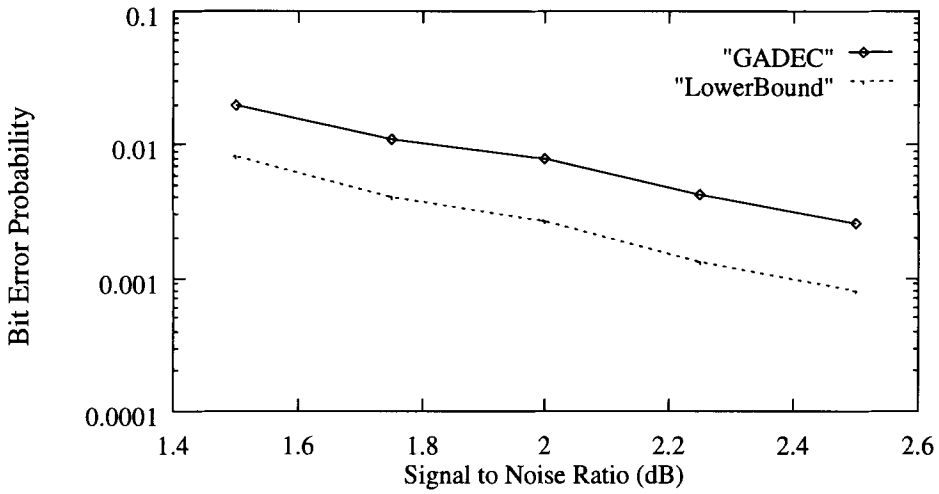

Figure 9: Bit Error Probability vs. Signal-to-noise Ratio, 50 generations: [104,52] Code. The lower bound represents errors that would necessarily be made by an optimal decoder and reflect cases where a codeword other than the transmitted codeword is closer in Euclidean distance to the received vector.

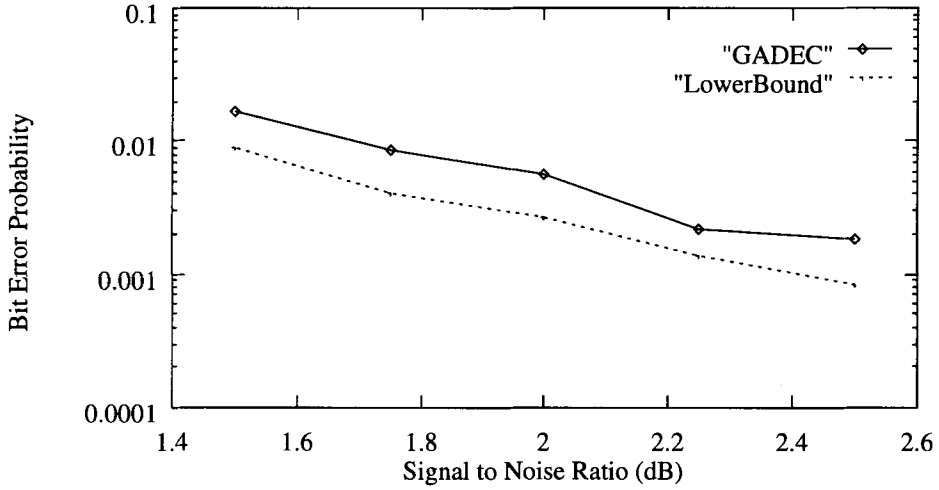

Figure 10: Bit Error Probability vs. Signal-to-noise Ratio, 100 generations: [104,52] Code.

error probability to the maximum likelihood decoding lower bound. P_b/MLD is as low as 1.65 to 2.2 after 100 generations, as given in Table 3. Increasing the number of generations results in further decrease in P_b/MLD, at the expense of more computation.

For a fixed bit error probability, it is possible to compute the difference in SNR between the lower bound maximum-likelihood decoding curve and the curve obtained from algorithm **GADEC**, as can be seen in Figure 9 and 10. This difference is atmost 0.55 dB after 50 generations of genetic search and reduces to atmost 0.35 dB after 100 generations.

3.2.3 Comparisons with other Operators and Algorithms

In Table 4 we present a comparison between results obtained using KNUX, uniform crossover (UX), and two point crossover (2PTX) in algorithm **GADEC**. In the case of two point crossover the algorithm was modified to produce two offspring. Simulation results show that KNUX is superior to UX and 2PTX by at least an order of magnitude.

We now compare our GA with the recently proposed A* based decoding algorithm, probably the fastest soft-decision decoding algorithm to date [3]. In the A* based algorithm, a linear code is represented as a trellis wherein each path represents a codeword [3]. The suboptimal version of algorithm A* restricts the list of nodes to be expanded for exploration based on a limit on memory size and prunes search paths which are estimated to contain the required solution with a probability less than threshold δ.

The bit error probability values obtained for the $[104, 52]$ code using a suboptimal version of algorithm A* are almost indistinguishable from those of algorithm **GADEC** after 100 generations of search. In addition, the dB difference for the A* algorithm is atmost 0.25 with $\delta = 0$ and 0.50 dB with $\delta = 0.25$ as compared with **GADEC** which has a dB difference of atmost 0.35 after 100 generations. The performance of 100 generations of algorithm **GADEC** is therefore seen to be better than that of A* with $\delta = 0.25$ and very close to that of A* with $\delta = 0.0$.

It is important to keep in mind that algorithm **GADEC** could be iterated further, until convergence, or perhaps reinitialized with new genetic material to continue the search even beyond 100 generations. This is a very significant advantage of genetic search techniques when one is willing to expend computation time for the sake of improved performance.

Another important advantage is the very low memory complexity of algorithm **GADEC**, which is $O(kN)$ as opposed to algorithm A* which, in the worst case has a memory complexity that is exponential in the dimension of the code, $O(n2^k)$.

Perhaps the most significant advantage of **GADEC** over the A* based approach is the fact that genetic algorithms are scalably parallel, suitable for implementation on a wide range of parallel architectures including massively parallel ones. There is also sufficient evidence to conclude that a distributed population version of algorithm **GADEC** would lead to better performance besides giving a good speedup [6], [10]. On the other hand an A* based algorithm is limited in speedup because it is necessary to compute the maximum node value at each level in the trellis before proceeding to the next one. Elsewhere [37], we provide a comparison of KNUX with systematic exhaustive search and pure random search to conclusively show that genetic recombination and selection indeed play an important role in obtaining solutions of high quality.

3.3 The Traveling Salesperson Problem

The traveling salesperson problem is a well studied combinatorial optimization problem which involves finding a minimal length tour of n cities that visits each city exactly once. Several researchers have attempted its solution with genetic algorithms [13], [33]. We show in this section how the KNUX and DKNUX philosophy of using problem-specific knowledge embodied in a good solution can be used to improve the performance of a crossover operator that has been used for this problem in the past. The choice of solution representation and crossover operator are clearly related. We represent tours as sequences of cities, where the ith element in a sequence is the ith city to be visited by the traveling salesperson. Grefenstette's crossover operator (referred to as 'GREF' here), can be described as follows [13]:

REPEAT

1. Randomly choose an unvisited city as the current city for the offspring tour.
2. Consider the four edges incident on the current city in the parents, ignoring edge direction. Define a probability distribution over these edges based on edge cost such that the probability associated with an edge incident to a previously visited city is 0. Each parental edge is assigned a probability $p_i = \frac{c_i}{\sum_{j=1}^{4} c_j}$ where $c_i = \frac{1}{(cost\ of\ edge\ i)}$.
3. Select an edge based on this distribution. (If none of the parental edges leads to an unvisited city, create an edge to a randomly chosen unvisited city.)

UNTIL all cities have been visited.

Operator KNUX_GREF is based on GREF, and associates probability p_i with a function of both the incident edge costs in parents and the best solution found so far. Let $\boldsymbol{b}$ denote the best solution found so far. The costs associated with edges are modified as follows:

$$c_i^{'} = c_i \text{ if edge } i \text{ is absent in } b$$

$$c_i^{'} = 2c_i \text{ if edge } i \text{ is present in } \boldsymbol{b}.$$

KNUX_GREF is identical to GREF except in step 2, where edge i is assigned the probability $\frac{p_i^{'}=c_i^{'}}{\sum_{j=1}^{4} c_j^{'}}$ instead of p_i as defined earlier. DKNUX_GREF is the dynamic variant of KNUX_GREF, that updates $p_i^{'}$ based on the best result found after each iteration.

3.3.1 Experimental Results

Experimental results obtained by implementing GREF and KNUX_GREF are shown in Figure 11. We show experiments on four symmetric traveling salesperson problems with 17, 21, 24 and 48 cities, respectively. Distance data for the problems used in given in the appendix. Figure 11 shows the results of the average over 5 runs of Algorithm DPGA configured as a hypercube of 16 subpopulations each of size 20. We find that operator KNUX_GREF is significantly better than operator GREF in all cases. Table 5 exhibits shows the minimum tour length obtained using operator GREF and operator KNUX_GREF. We see that KNUX_GREF is significantly better by this criterion as well.

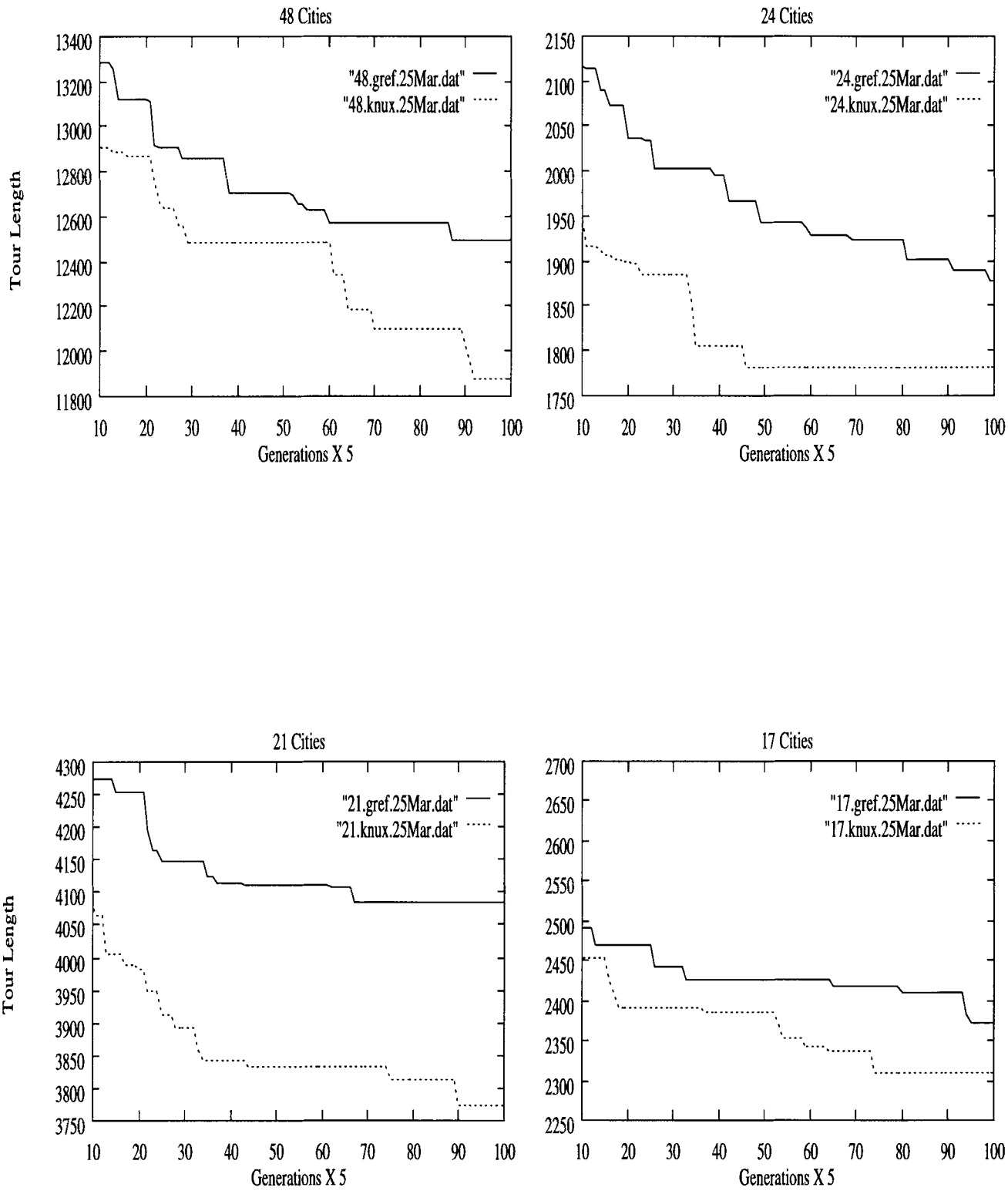

Figure 11: Tour Lengths in the TSP: A comparison between GREF and DKNUX_GREF.

Number of Cities	17	21	24	48
Minimum Tour Length Using GREF	2304	4017	1805	11997
Minimum Tour Length Using KNUX_GREF	2253	3691	1706	11609

Table 5: A Comparison of the Best Solutions found Using GREF and KNUX_GREF: Traveling Salesperson Problem.

4 Solution Improvement Using KNUX & DKNUX

KNUX and DKNUX can be used to improve the quality of solutions obtained using other algorithms. In this section we discuss the relative performance of KNUX and DKNUX when one member of the initial population is such a heuristic solution obtained using a greedy or hill-climbing approach. The quality of solutions obtained by KNUX depends on the quality of the heuristic estimate used to derive bias probabilities. It is therefore important to obtain a good, fast heuristic estimate of a solution. In the case of incremental graph partitioning, available solutions (partitions) provide a natural estimate.

For example, for the graph bipartitioning problem, the initial solution could be the one given by the IBP algorithm (see Appendix A), but after every N_0 generations, we may use the current "best" solution to generate the $\boldsymbol{P}$ matrix.

The philosophy is that the population and in particular the fittest surviving members of the population embody a wealth of knowledge about the search process. This knowledge can be exploited using the operators KNUX and DKNUX.

4.1 Starting with a Fast Heuristic Solution to Graph Partitioning

It is possible to seed the initial population of DPGA with a solution obtained through the Index Based Partitioner (Appendix A). This can then be improved on using DKNUX. The IBP algorithm uses coordinate information in graphs to establish a spatial proximity between graph nodes. In the absence of coordinate information, it is possible to assign pseudo-coordinates to the nodes of a graph by using a breadth-first search procedure as discussed in [22]. We have experimented with graphs that have 258, 167, 144 and 66 nodes which are illustrated in Appendix B, while initializing the initial population with a solution obtained using a fast heuristic such as the Index Based Partitioning method [29]. This heuristic utilizes the inherent proximity of graph nodes as given by their coordinates to determine an approximate partition of the graph.

We compare the performance of the crossover operators by comparing the best solution obtained and by also considering the number of generations it takes to achieve a solution of a particular quality as in Figure 13. Table 6 presents a comparison showing the quality of the best solution obtained using DKNUX with that of the solution obtained using RSB. The solutions obtained using DKNUX and KNUX exhibit a perfect load balance in all cases and are either better than or comparable to those obtained using RSB. Also, KNUX and DKNUX are far superior to two-point crossover as described in greater detail in [37].

Number of Partitions	2	4	8
167 Nodes Best Cut Using DKNUX	20	63	109
Cut Using RSB	20	59	120
144 Nodes Best Cut Using DKNUX	33	65	120
Cut Using RSB	36	78	119

Table 6: A Comparison of the Best Solutions found Using DKNUX and RSB: starting with a population initialized with an IBP (see Appendix A for IBP Algorithm) solution.

4.2 Starting With A Good Solution to Graph Partitioning

Recursive spectral bisection is a heuristic that has been widely reported to give very good solutions to the graph partitioning problem. This eigenvalue method does not use coordinate information and can thus be applied to graphs where such knowledge is not available. The disadvantage is that it is slower than the Index Based Partitioner even though it yields better solutions.

DKNUX can be exploited to refine the solutions given by RSB. Table 7 shows that DKNUX can yield a significant improvement over recursive spectral bisection solutions.

We have experimented with graphs of size 309, 279, 243, 213, 139 nodes. These graphs are given in Appendix B. Figure 12 shows the excellent performance of DKNUX in improving on recursive spectral bisection solutions. Figure 12 also shows that two-point crossover is able to make only an insignificant improvement.

Number of Partitions	2	4	8
139 Nodes Best Cut Using DKNUX	28	65	100
Cut Using RSB	30	69	113
213 Nodes Best Cut Using DKNUX	41	77	138
Cut Using RSB	41	82	151
243 Nodes Best Cut Using DKNUX	43	88	141
Cut Using RSB	47	95	154
279 Nodes Best Cut Using DKNUX	36	78	139
Cut Using RSB	37	88	155

Table 7: Improving the Solution found through Recursive Spectral Bisection.

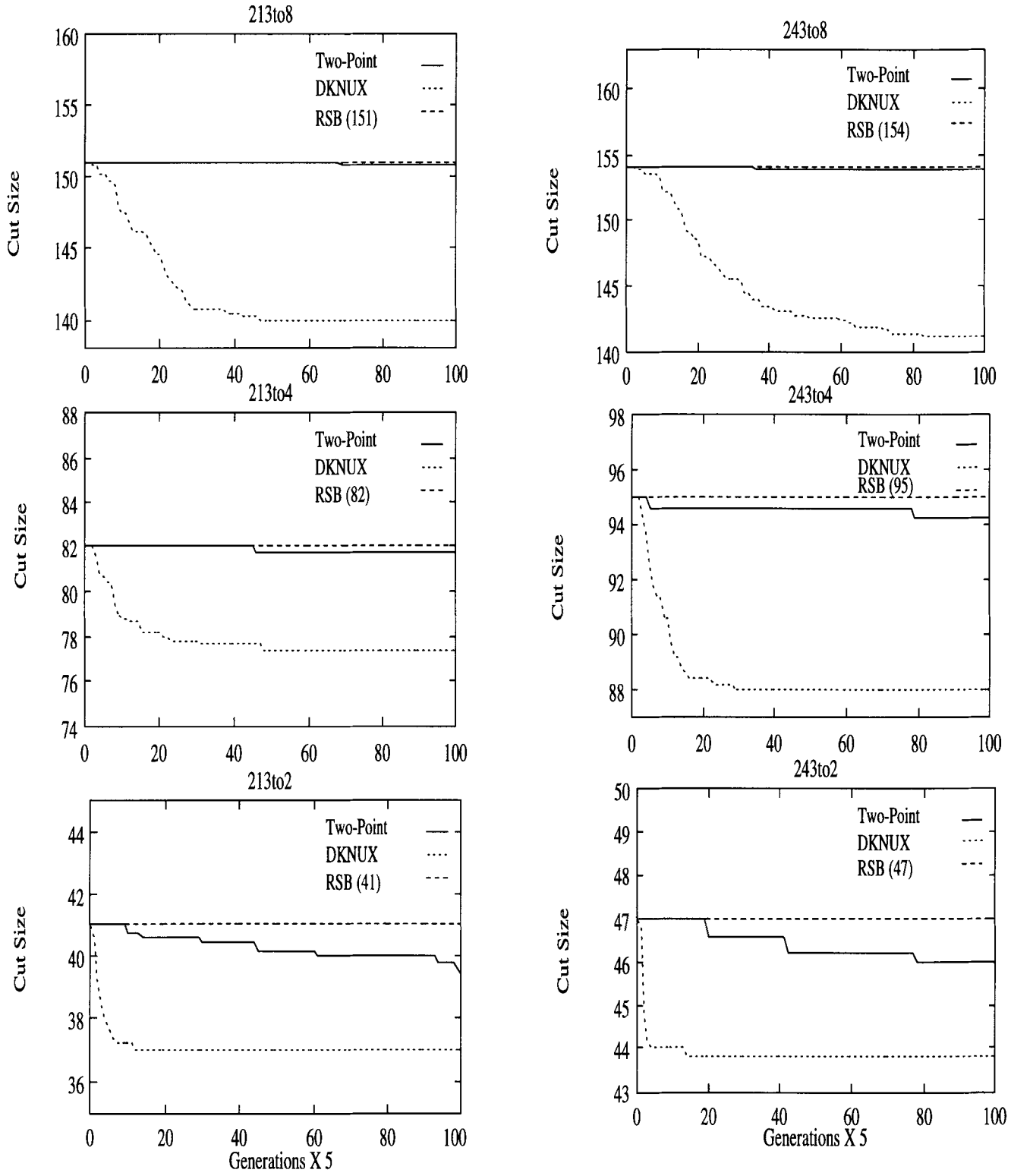

Figure 12: Partitioning 213 and 243 node graphs into 2, 4 and 8 partitions: The effect of operator DKNUX on improving solutions obtained through RSB

5 DKNUX for Incremental Optimization

Problems involving autonomously changing landscapes frequently arise when fitness is defined in terms of autonomous agents whose behavior can change, independent of the search activity. If typical GAs are applied to such problems, the strong pressures to converge results in a loss of diversity needed to respond to such changes. KNUX and DKNUX are useful in the context of changing fitness landscapes.

For a large class of irregular and adaptive data parallel applications such as adaptive meshes [21], the computational structure changes from one phase to another in an incremental fashion. In incremental graph partitioning problems, the partitioning of the graph needs to be updated as the graph changes over time; a small number of nodes or edges may be added or deleted at any given instant. A solution of the previous graph partitioning problem can be utilized to partition the updated graph, and the time required for such repartitioning would often be much less than the time required to applying a partitioning algorithm all over again to the entire updated graph.

In order to partition incrementally changing graphs we have experimented with adding a different number of nodes to graphs of sizes 249, 183, 119, 118 and 78.

We report results for two sets of incremental graph partitioning problems in Table 8 and Figure 13. In the first experiment, we start with a graph of 118 nodes and increment it by 21 nodes in a region chosen randomly within the graph. These three graphs are shown in the first row of Appendix B.

The incremented graphs are partitioned in 2, 4, and 8 partitions. The results are shown in Figure 13. DKNUX achieves excellent results in all three cases, and achieves in 100 generations a cut size that TPTX is unable to match even after 5000 generations. Similar results (see Figure 13) are obtained in the second incremental problem in which 41 nodes are added in a randomly chosen local area. For the 183 node graph incremented with 30 and 60 nodes, the graphs are shown in the second row of Appendix B. In [37] we show that DKNUX performs just as well in this case.

We find that the results representing an average over 5 runs of the algorithm (as given in Figure 13) are better than or within a few edges of those obtained by repartitioning the graph using the recursive spectral bisection heuristic. In addition, the best solution found using DKNUX is better than that obtained through RSB in all but one case.

6 Understanding KNUX

Most individuals in a GA have a transitory existence; hence properties of GA's are proved in terms of abstractions called schema that represent a collection of individuals (cf. section 1.2). Holland argues that schema that have been observed to have a higher average fitness are allocated an exponentially increasing number of trials. This strategy is optimal in the sense that viewing the problem as the k-armed bandit problem, one can show that the strategy maximizes payoff. The way the schema averages are estimated is implicit in the genetic algorithm and is done through a simultaneous sampling of several schemas while manipulating only a few chromosomes. KNUX and DKNUX consider some schemas as having a higher average fitness than others, with problem-dependent knowledge being used to determine what kinds of schemas

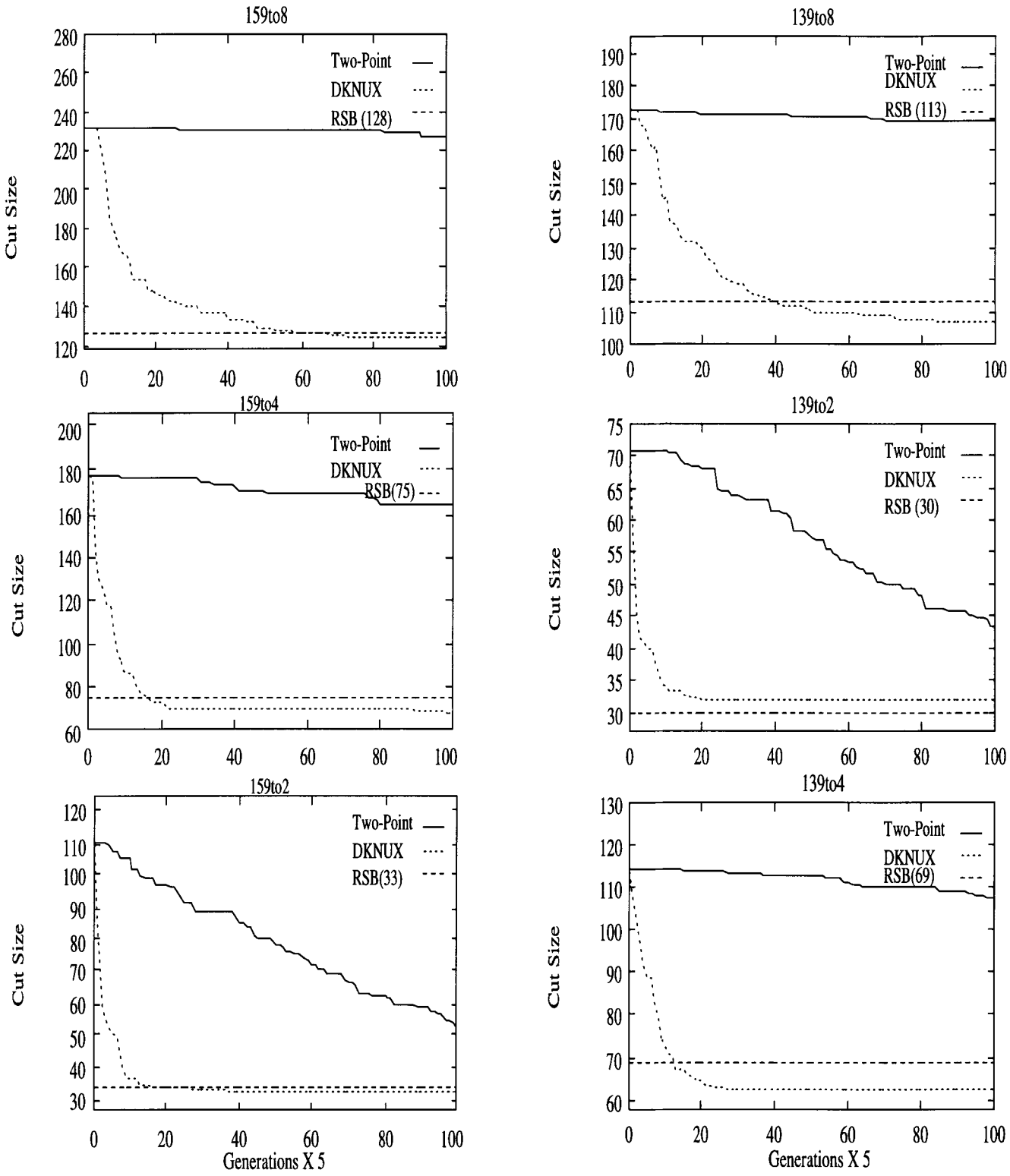

Figure 13: Partitioning a 118 node graph incremented by 21 nodes and 41 nodes; into 2, 4 and 8 partitions: A comparison between Two-Point, DKNUX and RSB

Number of Partitions	2	4	8
118 plus 21 Nodes			
Best Cut Using DKNUX	31	61	103
Cut Using RSB	30	69	113
118 plus 41 Nodes			
Best Cut Using DKNUX	31	66	120
Cut Using RSB	33	75	128
183 plus 30 Nodes			
Best Cut Using DKNUX	37	72	133
Cut Using RSB	41	82	151
183 plus 60 Nodes			
Best Cut Using DKNUX	44	83	160
Cut Using RSB	47	95	154

Table 8: A Comparison of the Best Solutions found Using DKNUX and RSB: Incremental Graph Partitioning.

are to be assigned a higher average fitness. Again, this is implicit in the genetic algorithm through the KNUX mechanism. DKNUX changes the estimate of schema fitness as the search progresses, abandoning some regions of the search space in favor of others. This is done utilizing the best performing individual and the information embodied in its structural makeup.

We provide an analysis of the schema theorem pertaining to KNUX for graph bipartitioning to better understand the forces driving this crossover operator. In this process we develop a variant of the schema theorem that leads to the "high-survivability building block" hypothesis.

6.1 Schema Disruption and Recombination

We begin by examining a traditional GA with individuals represented as binary strings, and with schema being strings ranging over the alphabet $\{1, 0, *\}$. The following notation is used:

- $o(S)$ = number of fixed positions in schema S.
- m = length of binary strings (chromosomes) in population.
- $P(t)$ is the population at iteration t, and $|P(t)| = N$ = population size.
- $\delta(S)$ = number of bits between the first and last fixed positions in schema S.
- $\psi(S,t)$ = number of strings in population $P(t)$ matched by schema S in generation t.
- $av(S,t)$ = average fitness of all strings in the population matched by schema S in generation t.
- $F(t)$ = Total fitness of all strings in population $P(t)$.
- p_c = Crossover Probability.

Assuming fitness proportionate selection it follows that:

$$\psi(S,t+1) = \psi(S,t)\frac{av(S,t)}{F(t)}N = \psi(S,t)\frac{av(S,t)}{F^{av}(t)},$$

where $F^{av}(t) = F(t)/N$. This reproductive growth equation says that the number of strings in the population matched by schema S grows as the ratio of the average fitness of schema S to the average fitness of the population. Accounting for the effects of one-point crossover and bit mutation by considering the disruptive effect of crossover and mutation on a schema S, we get:

$$\psi(S,t+1) \geq \psi(S,t)\frac{av(S,t)}{F^{av}(t)}(1 - \frac{P_c\delta(S)}{(m-1)})(1-p_m)^{o(S)}.$$

Above average schema with short defining length and low order are hence sampled at exponentially increasing rates.

This leads to the building block hypothesis, which says that a GA seeks near-optimal performance through the juxtaposition of short, low-order, high-performance schemata, called building blocks.

6.2 Schema Theorem for KNUX

The above analysis by Holland assumed a one-point crossover operator. We adapt the analysis for KNUX. Crossover is done by selecting a bit from either parent $\boldsymbol{a}$ or parent $\boldsymbol{b}$ with some probability. This probability is computed using problem specific knowledge and knowledge embodied in the best solution found so far.

We now consider the disruptive effect of KNUX on schema S. Let P_{r_i} = probability that the same bit as in schema-defined position is selected during crossover. In the case of graph bipartitioning, if the schema-defined position is 1, then $P_{r_i} = P(c_i = 1)$, and if the schema-defined position is 0, then $P_{r_i} = P(c_i = 0)$. P_{r_i} thus depends on the way the immediate neighbors are partitioned in the graph partitioning problem. P_{r_i} depends on the reliability of received vector coordinates in the soft-decision decoding problem. The probability that schema S survives KNUX is at least $\prod P_{r_i}$ where the product is taken over $o(S)$ positions. The crossover survival probability is hence at least $\prod P_{r_i}$ (where the product is taken over $o(S)$ terms corresponding to schema defined positions) instead of $(1 - P_c\delta(S)/(m-1))$. In our algorithms, utilizing KNUX, we use ranking selection instead of fitness proportionate selection [19]. Individuals are assigned a rank based on their fitness and we can interpret this rank as a variable or assigned fitness value [18]. Therefore, instead of using the fitness ratio $av(S,t)/F^{av}(t)$, we use $F(S,t) = \sum_{i \in S \cap P(t)} rank(i,t) / \sum_{i \in P(t)} rank(i,t)$, where $rank(i,t)$ = rank of individual i in generation t.

Hence, the reproductive growth equation takes the form:

$$\psi(S,t+1) \geq \psi(S,t)\ F(S,t)\ p_c\ \prod P_{r_i}\ (1-p_m)^{o(S)}$$

Clearly, $\psi(S,t)$ increases exponentially if $\prod P_{r_i}$ is large and if the contribution of the mutation term, $(1-p_m)^{o(S)}$, is small. This allows us to re-interpret the building block hypothesis as saying that, "above average, low-order, high-survivability schema are allocated an exponentially increasing number of trials."

It has been argued that survival and recombination abilities are mutually opposed criteria and tradeoffs are often needed [32]. We believe that KNUX alleviates this problem. The nonuniformity in KNUX allows the operator to be more effective at both schema recombination and survival, for the right sort of schema. KNUX can recombine some schema better than 2-point and parameterized uniform crossover, and disrupt some schema with a lower probability than 2-point and parameterized uniform crossover. This is unlike 2-point crossover which is less disruptive and less recombinative over all schema of a fixed order, and 0.5 uniform crossover which is more recombinative and more disruptive for all schema of a fixed order.

Parameterized uniform crossover does not differentiate between schema of a fixed order and 2-point crossover does not differentiate between schema of a fixed order and defining length. KNUX does differentiate between schema of a fixed order; building block recombination and disruption is independent of defining length. In a sense, this flexibility in recombining the right building blocks is what genetic algorithms are all about and to the best of our knowledge KNUX is the first recombination operator that exploits this mechanism. The question then arises: how are bias probabilities to be built? What schema are to be recombined more effectively than others and what schema are to be disrupted less frequently than others? The approach taken in KNUX is to let problem-specific knowledge guide this decision-making process. In addition, allowing bias probabilities to vary dynamically exploits both problem-specific information and information garnered about the search so far.

Example: Consider schema $*1********$ and schema $********01*$ that would recombine to give schema $*1****01*$. This would happen with probability approximately 0.7 for several variants of parametrized uniform crossover [32]. If p_i is the bias probability for the ith locus , schema $*1****01*$ would be reconstructed with a probability of at least $p_2(1-p_7)(1-p_8)$. Whenever $p_2 \times (1-p_7) \times (1-p_8)$ exceeds 0.7 (which would happen when $p_2 = 0.9$, $p_7 = 0.1$, $p_8 = 0.1$, for example), KNUX would recombine these schemas with a higher probability than both parametrized uniform and 2-point crossover.

Let us now consider the case of survival of a schema. Consider schema $1***11*****$, a third order schema whose survival probability is shown [32] to be about 0.9 under 0.1 uniform crossover, which is better (overall) than 2-point crossover. The survival probability of this schema under KNUX is at least $p_1 \times p_5 \times p_6$, which is an extremely conservative bound. Whenever this product exceeds 0.9, schema survival occurs with a higher probability in KNUX than in both 0.1 uniform and 2-point crossover.

6.3 Diploidy, Dominance, and DKNUX

Most advanced organisms are diploid, unlike most (artificial) genetic algorithms that use haploid representations. It has been claimed that the use of diploidy can help maintain the same amount of genetic diversity with much lower mutation rates than haploid representations [1, 38]. Furthermore, the dominance relationship among alleles can change when the environment changes (modeled by a change in the fitness function), so that the recessive alleles preserved in the diploid representation allow the population to adapt much more quickly to minor or major changes in the environment. However, diploid representations double the storage requirements (per-chromosome), increasing computational expense associated with each crossover and mutation step, and require additional representation in each chromosome to encode the dominance relationship as well as additional procedures that govern the changes in (adaptive) dominance

relationships when the environment changes (requiring increased manifestation of previously recessive alleles).

DKNUX may be viewed as a less expensive alternative that achieves some of the same goals as diploidy and adaptive dominance; dominance as well as the associated dominance-mutation rate are implicitly encoded in the reference vector used by DKNUX in each crossover step. Unlike biological organisms which must of necessity represent dominance relationships separately in every individual, computer implementations (using DKNUX) can encode these in a single reference vector accessible to all individuals, decreasing storage and computational requirements. Changes in the environment may result in previously fit individuals being downgraded, and these result in changes to the reference vector. Mutation is no longer the main driver in allowing the population to adapt to the new environment. Additional parameters that govern changes in dominance are no longer required. Furthermore, changes in the reference vector reflect changes in desirability of schema, rather than individual alleles as is the case with diploid representations that use a separate dominance parameter for each gene.

7 Conclusion

It is a truism that weak or generic methods are outperformed by specialized algorithms that utilize the maximum available knowledge about a given domain. For instance, a traditional GA would be outperformed by algorithms tailored to work well on the graph partitioning problem. Unfortunately, this leaves the user with very little direction about what to do when faced with a new problem. It would be ideal if the general algorithm adapts itself and "learns" about the environment, progressively improving in performance. This is one goal of KNUX, which may be considered a general-purpose method of incorporating knowledge specific to an application, with little user interaction.

In some applications, the ready availability of some knowledge or reasonably good solutions constitutes one reason to select symbolic computing systems (such as rule-based expert systems) over "soft computing" systems such as neural networks or GA's. KNUX provides a method of utilizing such knowledge in the context of GA's, with the ability of improving on solutions obtained by other methods.

DKNUX opens up the field of applying GA's to *Incremental Optimization* problems, characterized by a slow change in problem structure with time. In this respect, DKNUX also achieves some of the goals of diploid representations with adaptive dominance, with smaller computational requirements.

In general, heuristic search algorithms consist of move-generation followed by move-selection from any given state. GA's and other stochastic approaches often focus on improving the move-selection mechanism, after having chosen a fixed move-generation mechanism. KNUX differs from other modifications in making the move-generation process itself time-dependent. The same parents may give rise to different offspring at different moments in the evolutionary process, based on the past experience of the species.

KNUX and DKNUX bridge some of the gaps between evolutionary strategies and genetic algorithms in the use of "strategy" variables that determine genetic expression, and in maintaining a strong behavioral link between successive generations [36].

We have given simulation results showing that KNUX yields improvements (of orders of

magnitude in some cases) over two-point and uniform crossover, on three NP optimization problems: graph partitioning, soft-decision decoding of linear block codes, and the traveling salesperson problem. We have also applied KNUX to variants of the graph partitioning problem that cannot be solved easily using non-GA approaches, and to improve quality of solutions obtained using non-GA methods.

GA's have evolved considerably in recent years, with a large number of "species" of GA's taking hold. The usefulness of a new modification to GA's must hence be judged from its adaptability, i.e., the ease with which it can be combined with other improvements (both past and future) without producing a monster. We believe that the modification suggested by KNUX stands this test, these modifications are orthogonal to other changes in parameters of genetic algorithms, and can be pursued together with any other proposed improvements.

References

[1] J. H Holland, *Adaption in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, 1975.

[2] G. C. Clark and J. Bibb Cain, "Error Correcting Coding for Digital Communications", Plenum Press, 1988.

[3] Y. S. Han, "Efficient Soft Decision Algorithms for Linear Block Codes Using Algorithm A*," *Dissertation*, Technical Report SU-CIS-93-29, School of Computer and Information Science, Syracuse University, August 1993.

[4] K. H. Farell, L. D. Rudolph, and C. R. P Hartmann, "Decoding by local optimization," *IEEE TIT*, vol IT-29, No. 5, Sept. 1983.

[5] D. J. Taipale and M. B. Pursley, "An Improvement to Generalized Minimum Distance Decoding," *IEEE TIT*, vol 37, No 1, Jan. 1991.

[6] R. Tanese, "Distributed Genetic Algorithms," *Proc. of the 3rd ICGA*, 1989, pp. 434–439.

[7] D. E. Goldberg, "Genetic Algorithms and Walsh Polynomials: Part I, A gentle Introduction," *Complex Systems*, 3, 1989, pp. 129–152.

[8] D. E. Goldberg, "Genetic Algorithms and Walsh Functions: Part II, Deception and its Analysis," *Complex Systems*, 3, 1989, pp. 153–171.

[9] M. G. Schleuter, "An asynchronous parallel genetic optimization strategy," *Proc. of the 3rd ICGA*, 1989, pp. 422–428.

[10] H. Muhlenbein, "Parallel genetic algorithms, population genetics and combinatorial optimization," *Proc. of the 3rd ICGA*, 1989, pp. 416–422.

[11] R. Collins and D. Jefferson, "Selection in massively parallel genetic algorithms," *Proc. of the 4th ICGA*, 1991, pp. 249–256.

[12] J. P. Cohoon, W. N. Martin, and D. S. Richards "A multi-population genetic algorithm for solving the k-partition problem on hypercubes," *Proc. of the 4th ICGA*, 1991, pp. 244–248.

[13] J. J. Grefenstette, "Incorporating problem specific knowledge into genetic algorithms," *Genetic Algorithms and Simulated Annealing*, L. Davis and Morgan Kaufmann, eds., 1987.

[14] G. Syswerda, "Uniform Crossover in Genetic Algortihms", *Proc. of the 3rd ICGA*, 1989, pp. 2-9.

[15] J. D. Schaffer, L. J. Eschelman and D. Offut, "Spurious Correlations and Premature Convergence in Genetic Algorithms", *Foundations of Genetic Algorithms*, Morgan Kaufmann, 1991, pp. 102-115.

[16] L. Eschelman, "The CHC Adaptive Search Algorithm: How to Have Safe Search When Engaging in Nontraditional Genetic Recombination", *Foundations of Genetic Algorithms*, Morgan Kaufmann, 1991, pp. 265-284.

[17] Z. Michalewicz, "Genetic Algorithms + Data Structures = Evolution Programs", Springer Verlag, Berlin, 1992.

[18] D. Whitley, "The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is best", *Proc. of the 3rd ICGA*, 1989, pp 116-121.

[19] J. E. Baker, "Reducing bias and inefficiency in the selection algorithm," *Proc. of an International Conference on Genetic Algorithms and their Applications*, Erlbaum, 1985.

[20] L. Booker, "Improving search in genetic algorithms," *Genetic Algorithms and Simulated Annealing*, L. Davis, Morgan Kaufmann, 1987.

[21] A. Choudhary, G. C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and J. Saltz, "Software support for irregular and loosely synchronous problems," In *Proceedings of the Conference on High Performance Computing for Flight Vehicles*, 1992.

[22] Yeh-Ching Chung, Sanjay Ranka, "Mapping finite element graphs on hypercubes," *The Journal of Supercomputing*, 6, 1992, pp. 257–282.

[23] G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, *Solving Problems on Concurrent Processors*, Prentice Hall, 1988.

[24] M. R. Garey and D. S. Johnson, *Computers and Intractability* San Francisco, CA, Freeman, (1979), pp. 209–210.

[25] D. R. Jones and M. A. Beltramo, "Solving partitioning problems with genetic algorithms," *Proc. of the 4th ICGA*, 1991, pp. 442–450.

[26] Gregor von Laszewski, "Intelligent structural operators for the k-way graph partitioning problem," *Proc. of the 4th ICGA*, 1991, pp. 45–52.

[27] H. S. Maini, K. G. Mehrotra, C. K. Mohan, S. Ranka, "Genetic algorithms for soft-decision decoding of linear block codes," Technical Report SU-CIS-93-25, School of Computer and Information Science, Syracuse University, 1993. To appear in *Evolutionary Computation.*

[28] Nashat Mansour, *Physical Optimization Algorithms for Mapping Data to Distributed-Memory Multiprocessors*, Ph.D. Thesis, School of Computer and Information Science, Syracuse University, 1992.

[29] Chao-Wei Ou, Sanjay Ranka, and Geoffrey Fox, "Fast mapping and remapping algorithm for irregular and adaptive problems," Technical Report, July 1993.

[30] W. M. Spears and K. A. DeJong, "An analysis of multipoint crossover," In G. Rawlins (ed.) *Foundations of Genetic Algorithms*, San Mateo, CA: Morgan Kaufmann, 1991.

[31] J. H. Holland "Genetic Algorithms and The Optimal Allocation of Trials," SIAM Journal of Computing, Vol. 2, No. 2, June 1973.

[32] W. M. Spears, K. A. De Jong, "On the Virtues of Parameterized Uniform Crossover", Proc. of the 4th ICGA, 1991, pp230-242.

[33] J. Y. Suh, D. V. Gucht, "Incorporating Heuristic Information into Genetic Search," *Genetic Algorithms and Their Applications: Proceeding s of the Second International Conference on Genetic Algorithms.* ed J. J Grefenstette, pp100-107.

[34] Sushil J. Louis, G. Rawlins, "Designer Genetic Algorithms: Genetic Algorithms in Structure Design," *Proc. of the 4th ICGA*, 1991, pp 53-60.

[35] David B. Fogel, "On the Philosophical Differences between Evolutionary Algorithms and Genetic Algorithms," *Proc. of the 2nd Annual Conference on Evolutionary Programming*, Feb. 1993, La Jolla, CA.

[36] T. Back, G. Rudolph, H. P. Schewefel, "Evolutionary Programming and Evolution Strategies: Similarities and Differences," *Proc. of the 2nd Annual Conference on Evolutionary Programming*, Feb, 1993, La Jolla, CA.

[37] H. S. Maini, K. G. Mehrotra, C. Mohan, S. Ranka, "Genetic Algorithms for Graph Partitioning and Incremental Graph Partitioning," Technical Report SU-CIS-94, March 1994.

[38] R. E. Smith, D. E. Goldberg, "Diploidy and Dominance in Artificial Genetic Search," Complex Systems, Vol. 6, 1992, pp. 251-285.

Appendix A: Index-Based Partitioning (IBP) Algorithm

Index-based algorithms to partition graphs have been described in [29]. An IBP algorithm includes three phases— indexing, sorting, and coloring. The indexing scheme is based on converting an N-dimensional co-ordinate into a one-dimensional index such that proximity in the multi-dimensional space is maintained. Row-major indexing and shuffled row-major indexing are two of the several ways of indexing pixels in a two-dimensional grid. These two indexing schemes are shown in Figure 14 for a graph in which the set of vertices are arranged in a grid of size 8×8.

```
00 01 02 03 04 05 06 07    00 01 04 05 16 17 20 21
08 09 10 11 12 13 14 15    02 03 06 07 18 19 22 23
16 17 18 19 20 21 22 23    08 09 12 13 24 25 28 29
24 25 26 27 28 29 30 31    10 11 14 15 26 27 30 31
32 33 34 35 36 37 38 39    32 33 36 37 48 49 52 53
40 41 42 43 44 45 46 47    34 35 38 39 50 51 54 55
48 49 50 51 52 53 54 55    40 41 44 45 56 57 60 61
56 57 58 59 60 61 62 63    42 43 46 47 58 59 62 63

        (a)                        (b)
```

Figure 14: (a) Row-Major and (b) Shuffled Row-Major Indexing for an 8×8 image

A simple example of interleaving indices is as follows. Suppose $index_1 = 001$, $index_2 = 010$, and $index_3 = 110$. Then the interleaved index would be 001011100. In the above case the number of bits in each dimension are equal. This could easily be generalized to cases when the sizes are different. For example if $index_1 = 101$, $index_2 = 01$, and $index_3 = 0$, then the interleaved index would be 100110. This is done by choosing bits (right to left) of each of the dimensions one by one, starting from dimension 3. When the bits of a particular dimension are no longer available, that dimension is not considered.

After indexing is done, an efficient sorting algorithm can be applied to sort these vertices according to their indices. Finally, this sorted list is divided into P equal sublists.

Appendix B: Graphs Partitioned

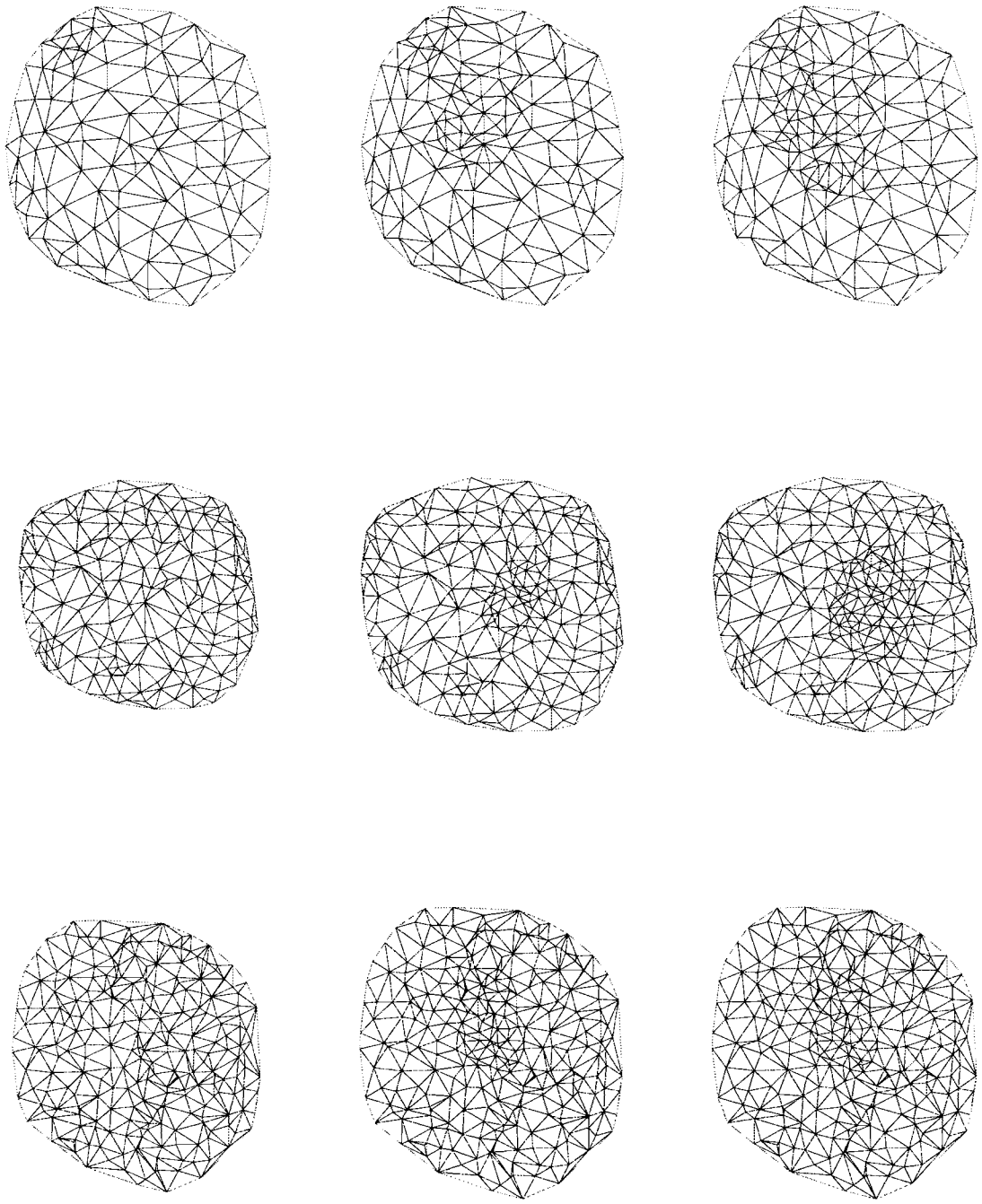

Figure 15: Irregular Graphs of 118, 139, 159, 183, 213, 243, 249, 279 and 309 nodes in a row major order (first row followed by second, followed by third).

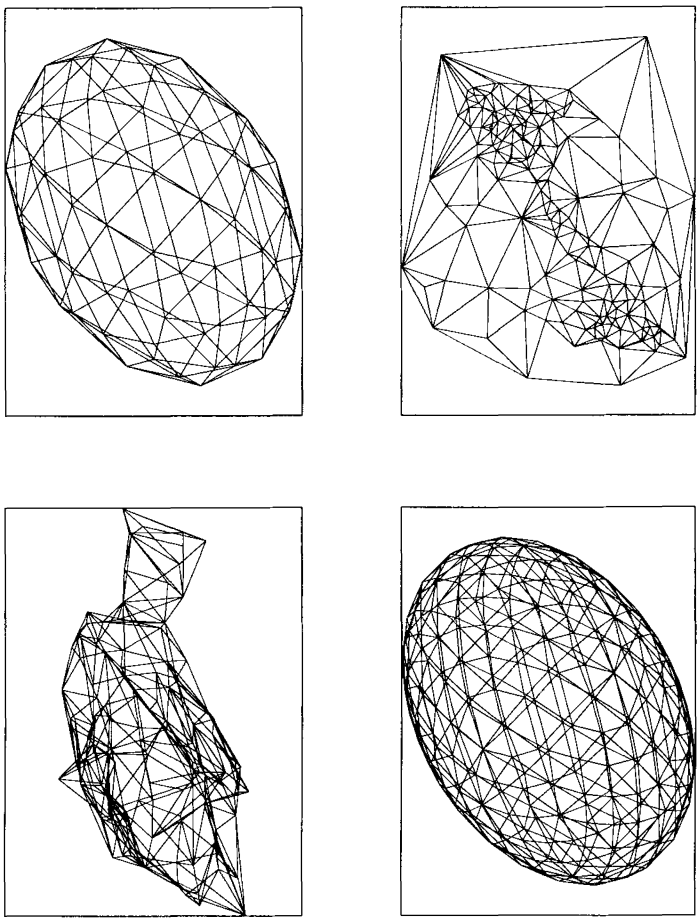

Figure 16: Graphs of 66, 167, 144 and 258 nodes in a row major order